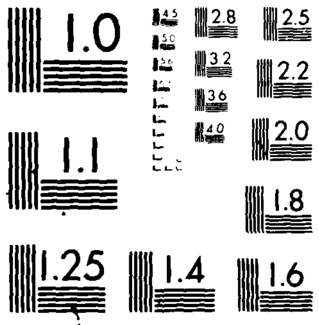
1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

LEVEL II

12

ISI/RR-81-89
March 1981

Carl A. Sunshine

Formal Modeling of Communication Protocols

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA ISI 4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 ISI/RR-81-89	2. GOVT ACCESSION NO. AD-A098	3. RECIPIENT'S CATALOG NUMBER 577
4. TITLE ~~(and Subtitle)~~ 6 Formal Modeling of Communication Protocols,		5. TYPE OF REPORT & PERIOD COVERED 9 Research rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Carl A./Sunshine		8. CONTRACT OR GRANT NUMBER(s) 15 DAHC15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 12 32
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209 11		12. REPORT DATE March 1981
		13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) -------		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

----

18. SUPPLEMENTARY NOTES

----

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstract machine, protocol, specification, state exploration, symbolic execution, temporal logic, verification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED 407933

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Interest in more rigorous definition and analysis of communication protocols is increasing. This report surveys the current state of the art in protocol specification and verification. Methods for specification such as abstract machines, Petri nets, formal languages, abstract data types, and programs are described and compared. Verification methods including state exploration, symbolic execution, structural induction, and program proof are discussed. Work is progressing rapidly in many of these areas, and no clearly superior method has emerged yet. At least in the area of specification, some of these methods are ready for use by a wider community of protocol designers and users.

ISI/RR-81-89
March 1981

Carl A. Sunshine

Formal Modeling of Communication Protocols

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research is supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

CONTENTS

FIGURES

1. INTRODUCTION

As computer networks proliferate, the design of properly functioning communication procedures or *protocols* becomes ever more important. Traditional methods of informal narrative specifications and ad hoc validation have demonstrated their shortcomings as protocol "bugs" crop up. Problems of ambiguous and incomplete specifications are particularly severe for the ever-growing number of protocol standards that must be implemented by a wide community of users with diverse equipment.

This report surveys recent progress in making the protocol design process more rigorous. Section 2 clarifies the meaning of protocol specification and verification. Section 3 describes various specification methods, and section 4 presents verification methods.

The work reported here was performed as part of the Internet Concepts Research project sponsored by the Defense Advanced Research Projects Agency. The overall goals of this project are to develop a complete set of protocols to support diverse user applications in a multinetwork environment. One aspect of this work concerns development of more rigorous protocol specification and analysis methods to ensure the correctness of system designs and implementations. This report presents a survey of relevant formal methods. Future reports will include more detailed studies of particular methods.

2. MEANING OF SPECIFICATION AND VERIFICATION

We assume that the communication architecture of a distributed system is structured as a hierarchy of different protocol layers. Each layer provides a particular set of *services* to its users above. From their viewpoint, the layer may be seen as a "black box" or machine which allows a certain set of interactions with other users (see Fig. 2.1). A user is concerned with the nature of the service provided, but not with how the protocol manages to provide it.

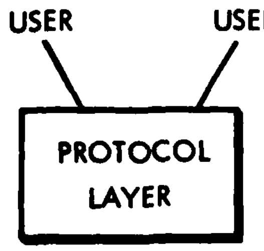

Figure 2.1. User view of protocol layer

This description of the input/output behavior of the protocol layer constitutes a *service specification* of the protocol. It should be "abstract" in the sense that it describes the types of commands and their effects, but leaves open the exact format and mechanisms for conveying them (e.g., procedure calls, system calls, interrupts).

Although the internal structure of a protocol layer is irrelevant to the user, the protocol designer must be concerned with it. In a network environment with physically separated users, a protocol layer must be implemented in a distributed fashion, with *entities* (processes or modules) local to each user communicating among one another via the services of the lower layer (see Fig. 2.2). These entities may be considered "black boxes" in their own right, and the rules by which they interact with each other (and their users) to provide the layer's service constitutes the actual *protocol*. Hence a protocol specification must describe the operation of each entity within a layer in response to commands from its users, messages from the other entities (via the lower layer service), and internally initiated actions (e.g., timeouts).

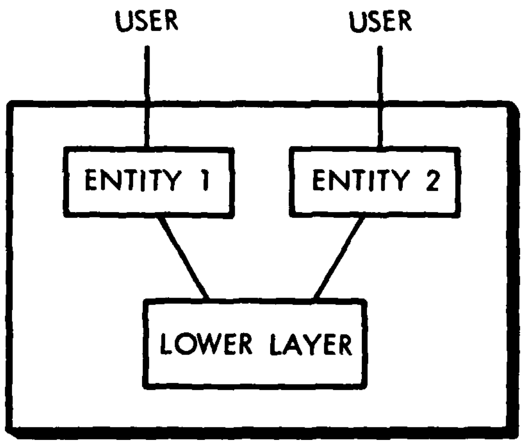

Figure 2.2. Internal structure of protocol layer

A protocol specification is a refinement or distributed "implementation" of its service specification in the sense that it partly defines how the service is provided (i.e., by a set of cooperating entities). This "implementation" of the service is what is usually meant by the design of a protocol layer. The protocol specification should define each entity to the degree necessary to ensure compatibility with the other entities of the layer, but no further. Each entity remains to be implemented in the more conventional sense of that term, typically by coding in a particular programming language.

Verification is essentially a demonstration that an object meets its specifications. From the above discussion, we see that the service and the protocol are the two major items requiring specification for a protocol layer. Hence there are two basic verification problems that must be addressed: (1) the protocol's *design* must be verified by analyzing the possible interactions of the entities of the layer, each functioning according to its (abstract) protocol specification and communicating through the underlying layer's service, to see whether this combined operation satisfies the layer's service specification; and (2) the *implementation* of each protocol entity must be verified against its abstract protocol specification.

The term "protocol verification" is usually intended to mean this first design verification problem. Because protocols are inherently systems of concurrent independent entities interacting via (possibly unreliable) exchange of messages, verification of protocol designs takes on a characteristic communication-oriented flavor. Implementation of each entity, on the other hand, is usually done by "ordinary" programming techniques, and hence represents a more common (but by no means trivial) program verification problem that has received less attention from protocol verifiers.

It is only recently that the need for complete service specifications has been realized. Hence much of the work to date has attempted to verify plausible but rather ad hoc sets of properties. There are also a set of desirable properties common to most protocols (and other systems) including:

- Freedom from deadlock, or arrival at some desired final state(s);
- Completeness of the protocol in handling all situations that may arise during execution;
- Progress, or the absence of cycles where no "useful" work is done;
- Stability, meaning that the protocol will return to a "normal" mode of operation after an abnormal condition occurs.

These properties may be viewed as implicit requirements on any protocol, and may be verified in the absence of an explicit service specification.

3. SPECIFICATION METHODS

Protocol specification methods have traditionally developed from two viewpoints: state machines and programs. As we shall see, there is really no fundamental difference between these approaches, and a variety of elaborations that span the gap between them have been proposed. But these two categories still provide a useful starting point for discussion.

Program specifications are motivated by the observation that protocols are merely a particular class of information processing procedures, and that various programming languages provide a convenient means for describing such procedures. Because they typically involve multiple concurrent processes, protocols do present some special challenges, but programming languages catering to such problems have been developed.

The state machine model is motivated by the observation that protocols may be conveniently viewed as rules specifying the responses or outputs of a protocol "machine" to each command or input. The response to an input typically depends on the type of that input and on the history of past inputs, or the *state* of the machine. The key components of these models are definitions of: (1) a set of commands or inputs, (2) one or more state variables, (3) a *transition function* from (command x state) → state, and (4) an initial state (initial values for all the state variables). Each command to the system causes the system to enter a new state which is based on the current state and the nature of the command.

Within these basic guidelines, there are a number of possible variations. State variables may be defined as value-returning functions. The commands may have parameters. The effects of commands may be made visible to the outside world (users) by defining some of the state variables to be visible, or by producing explicit outputs as additional effects of an operation. "Exceptional" conditions may be specified where a given command has no effect on the state of the system except to produce an error indication or output to the invoking user. Generally, the commands are considered atomic operations that are processed sequentially, but concurrent commands are allowed in some models. In the following sections we shall discuss a number of the major combinations of these features that have been proposed.

Finite State Automata

Finite state automata are one of the simplest types of state machine model because they have only a single state variable (*the* state) which takes on a relatively small range of values. For application to protocols, the states typically correspond to different "phases" of operation, e.g., idle, call setup, data transfer, interrupt, call clearing.

FSA may be specified formally by giving their input and output sets, state set, initial state, and transition functions for next state and output. A great deal of work has been done on their formal properties. One virtue of FSA is that they may be written graphically to facilitate understanding, with circles representing states, and arcs representing transitions. Each arc is labeled with the input which causes the transition. Outputs produced are also written on the arcs if needed. Figure 3.1 gives an example of an FSA for a very simple connection establishment protocol.

FSA may also be presented in tabular form with states as rows and inputs as columns (or vice versa). The resulting matrix has the appropriate next state and output entered for each cell. Or the information may be presented linearly by rows (for each state, give the effects of each input), or by columns (for each event, give the results for each state), as proves most convenient.

While "pure" FSA models have been used to define the control features of some protocols such as CCITT X.21 [WeZa 78], their limitation of a single state variable quickly becomes apparent when protocol features such as sequence numbers, message texts, and timers must be included in the specification. For example, a separate state would be needed for each possible value of a pending

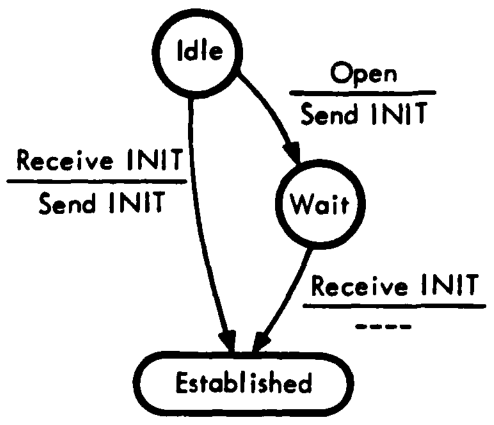

Figure 3.1. Finite State Automaton for connection establishment

message and/or sequence number in a pure FSA model. Hence extension of the model to include a number of different type state variables becomes desirable, as discussed next.

Abstract Machine Model

We shall use the term *abstract machine* to represent the extension of an FSA obtained by allowing multiple state variables of various types. The major difference is that the "state" now becomes a vector of these variables, and the transition functions giving the next state and the output become correspondingly more complex. If the data types of the state variables are unbounded (e.g., a queue), the model may not even have a finite number of states.

An abstract machine model for a protocol often has a distinguished state variable (called "the state") which still serves to identify the major phases of the protocol. In these cases, the same graphical model as for FSA may be used, with circles representing the major states, and arcs the transitions between states. Each arc is labeled with the event causing the transition (which typically includes conditions on the rest of the state vector and any parameters of the event), with outputs produced, and with any changes to the rest of the state vector. The tabular specification must be similarly augmented.

Many people have proposed particular forms of such abstract machine models for specifying protocols [LeMo 73, Boch 78, DaBr 78, SuDa 78, Piat80, Tenn80, Dick 80, and SDCo 80]. While differing in details of syntax, the proposals may be considered equivalent in their basic intention.

A useful elaboration appearing in several methods is the ability to define "submachines" which give the details of operation within what appears as a single "compound" state of a higher level machine [SDCo 80]. Entry into the parent level's compound state causes activation of the appropriate submachine at its initial state, and any inputs not handled at the parent level are passed to the submachine. Exit from the parent state terminates the submachine no matter which substate it is in. This corresponds roughly to the subroutine notion in programming, and has been called "hierarchical dependence" by [Boch 78].

Two formal specification languages have been developed expressly for defining such abstract machines: Special [RLSi 79] and Ina Jo [Loca 80]. Special is based on the work of Parnas [Parn 72] where the state is represented by a set of value-returning functions (*Vfunctions*), and the commands by a set of state- modifying functions (*Ofunctions*) which set the values subsequently returned by some Vfunctions. The Ofunctions may also have a test for exception conditions which cause an error return instead of their normal effects. In Special, there is no explicit output produced, but some of the Vfunctions are meant to be "observable" to the users, while others are not visible to the users and represent purely internal state information.

The state variables in Special may include basic types such as integer, record, sequence, set, and boolean, and also user-defined types that are defined independently as other abstract machines. The effects section of Ofunctions may then call on the Ofunctions of the more primitive types. The Ofunctions may also include *delay until* constructs which cause an operation to wait until the specified property of the state is true before it is effected. A service specification for a simple single-message-at-a-time data transfer protocol is given in Figure 3.2.

In Ina Jo, the state is defined directly as a number of variables of different types, although there is no facility for using separately defined machines as additional types. The effects of commands are given in *transforms* which specify the new values of any changed state variables as a function of their old values and the parameters of the command. Ina Jo also allows the specification of properties that must be maintained by the transforms, but further consideration of these is postponed until our discussion of verification.

A difficulty with abstract machine models stems from their purely passive nature: they define the effects of operations which are invoked, but do not include any notion of an active agent which may cause events or invoke operations on its own initiative. For example, the effects of a Retransmit operation may be defined, but not the fact that the operation is to be invoked whenever a certain time elapses. This shortcoming may be partially avoided by interpreting some events as "external" or callable by a user of the machine, and other events as "internal" or called by some sort of daemon process associated with the protocol machine itself. Understandably, this problem is more serious when the issue is liveness rather than safety.

```
MODULE ProtocolService

TYPES
   msg: ...
   state: SET {Empty, Full}

FUNCTIONS

VFUN Buf() → msg m
   HIDDEN

VFUN State() → state s
   INITIALLY s = Empty

OFUN Send(msg m)
   EXCEPTION State = Full
   EFFECTS
      'Buf = m  [' means new value of]
      'State = Full

OVFUN Receive() → msg m
   EXCEPTION State = Empty
   EFFECTS
      m = Buf
      'State = Empty

END MODULE ProtocolService
```

Figure 3.2. Protocol service in Special

Formal Languages

Formal languages and the grammars which define them are another variation of state machine type models. If we view the sequence of inputs and outputs of a protocol machine as sentences of a formal language, then we can define the formal grammar which would produce all valid sentences. There is a well-known correspondence between such grammars and various sorts of automata which will recognize valid sentences of the language.

For certain types of grammar, this correspondence is very apparent. For each state of the state machine model, we define a nonterminal symbol in the grammar. The inputs and outputs correspond to terminal symbols. For each transition from state S to state T with input i and output o (which may be null), we add a production to the grammar, S:: = ioT. The initial state of the machine becomes the starting symbol of the grammar. Figure 3.3 shows an example of this type of specification for the "Alternating Bit" protocol. Of course it is possible to simplify the resulting grammar by substituting later productions for the nonterminals in earlier productions, thereby making the omitted states implicit in the succession of terminal symbols.

```
S0::= <get new message>  C0
C0::= Send0  W0
W0::= ReceiveAck1  S1              S0::= ReceiveMsg0  <give message to user>  A1
         ReceiveAck0  C0                     ReceiveMsg1  A0
         Timeout  C0                         ReceiveMsgError  A0
                                   A0::= SendAck0  S0
S1::= <get new message>  C1
C1::= Send1  W1                    S1::= ReceiveMsg1  <give message to user>  A0
W1::= ReceiveAck0  S0                        ReceiveMsg0  A1
         ReceiveAck1  C1                     ReceiveMsgError  A1
         Timeout  C1                         A1::= SendAck1  S1

         (a) Sender                          (b) Receiver
```

Figure 3.3. Alternating bit protocol in formal grammar

One disadvantage of formal grammar models is apparent from the example. They do not distinguish between inputs and outputs (although the names of the symbols may be chosen to informally indicate this). The usual interpretation of grammars as defining either a recognizer for the language or a producer makes it difficult to view them as both (recognizer of inputs and producer of outputs) as necessary for protocols.

Formal language models suffer from the same limitations as FSA in representing the text of messages or the values of sequence numbers. An indexing scheme for defining groups of productions parameterized by a sequence number has been proposed by Harangozo [Hara 77]. While maintaining a compact notation. this effectively multiplies the number of productions (or states).

An advantage of grammars is that the same formalism may be used to represent the format of messages exchanged. Thus there may be an "action grammar" for the allowed sequences of events, and a "message grammar" for the format of fields in each event [TeLi 78]. The productions of the latter may be substituted for the terminals of the former, yielding a more detailed description of the interactions.

Sequencing Expressions

Sequencing expressions are an attempt to specify the sequence of protocol operations directly. The basic operators for building up such expressions are repetition, follows, and alternatives. A simple form are the well-known *regular expressions* that correspond to FSA (and type 3 formal grammars). As with grammars, there is no formal distinction between inputs and outputs in the sequencing expressions (but naming conventions may be used). However, there is no longer any explicit state that must be invented and specified. Instead, the current location in the expression is an implicit state.

Schindler has proposed a number of extensions to facilitate protocol specification [Schi 80]. The overall expression may be broken into several *blocks*, each block functioning much as a nonterminal in a grammar. Each term in the expression may have a *rejection predicate* which causes an otherwise allowed operation to be deemed invalid if false. These predicates may refer to the parameter values of the current and previous operations and to the number of operations of a given type previously executed (successfully).

Each block may also have several *exit* blocks specified, so that an operation that does not match within the currently active block may match (and cause entry to) an exit block. These are intended to specify the handling of "abnormal" events such as a Disconnect during data transfer. Thus, the exit block construct serves to define an alternative which "distributes over" all the items in the current block. Operations that do not match the current pointer or any exit blocks are "rejected" or ignored with no effect on the system.

An example (taken from [Schi 80]) of this type specification for the X.25 Level 3 protocol is given in Figure 3.4. Note that only part of the data transfer constraints are represented in the sequencing expression here (DIT block), which must be augmented by an abstract machine type specification for the data tr^nsfer operations (not shown here for brevity). The complete constraints could be captured in the sequencing expressions, but we have chosen the simpler example for ease of understanding. Thus the explicit state machine and sequence expression methods can be mixed in whatever proportion seems clearest.

The original work by Schindler on sequencing expressions focused on the interactions between protocol entities and omitted user input/output (from/to the level above). Reference [SFAI 80] describes recent extensions to include user interfaces, with particular emphasis on service specifications. Bochmann has also proposed a sequence notation for specifying protocol services [Boch 80].

```
PSE X.25/L3 := { CSU → {DIT* → RES}* → CLR };

CSU := { (s.cr#T21# → r.crconf (r.cr → r.crconf))
          (r.cr → s.crconf) },
        X2 := CLR;

DIT := { s.d r.d s.rr r.rr s.rnr r.rnr s.i r.i s.iconf r.iconf },
        RP := 'nx(s.i) = nx(r.iconf) + 1  AND
               'nx(s.iconf) = nx(r.i)  AND
                's.d ==> r.rnr → r.rr  RPend,
        DP := 'nx(r.i) = nx(s.iconf) + 1  AND
                'nx(r.iconf) = nx(s.i)  DPend,
        X1 := RES, X2 := CLR;

RES := { {r.d r.rr r.rnr r.i r.iconf r.resconf}*
           → { (r.res → s.resconf)
                ( (s.res → {r.d r.rr r.rnr r.i r.iconf}* )#T22#
                   → r.resconf r.res ) } },
        X1 := RES, X2 := CLR;

CLR := { {r.≠cl}*
        → (r.cl → s.clconf)
           ( (s.cl → {r.≠clconf≠cl}* )#T23# → r.clconf r.cl) },
        X2 := CLR;

END.MODULE
```

Notes: s.XX and r.XX mean send and receive message type XX respectively. Message types are from X.25 (e.g., data, call request, receiver ready, reset, interrupt). The term nx(XX) means the number of type XX messages accepted so far. Terms separated by spaces within brackets are alternatives. → means followed by. * means arbitrary repetition. For more details on syntax, see references cited in the text.

Figure 3.4. CCITT X.25 as protocol sequence expression

Petri Nets

Petri nets and the related UCLA graph model [Post 74, RaEs80] are other graphical formalisms used to specify protocols. The *places* in Petri nets correspond to the states of a protocol machine, but also are used to represent inputs and outputs. Figure 3.5 (from [Merl 79]) shows the Alternating Bit protocol as a Petri net.

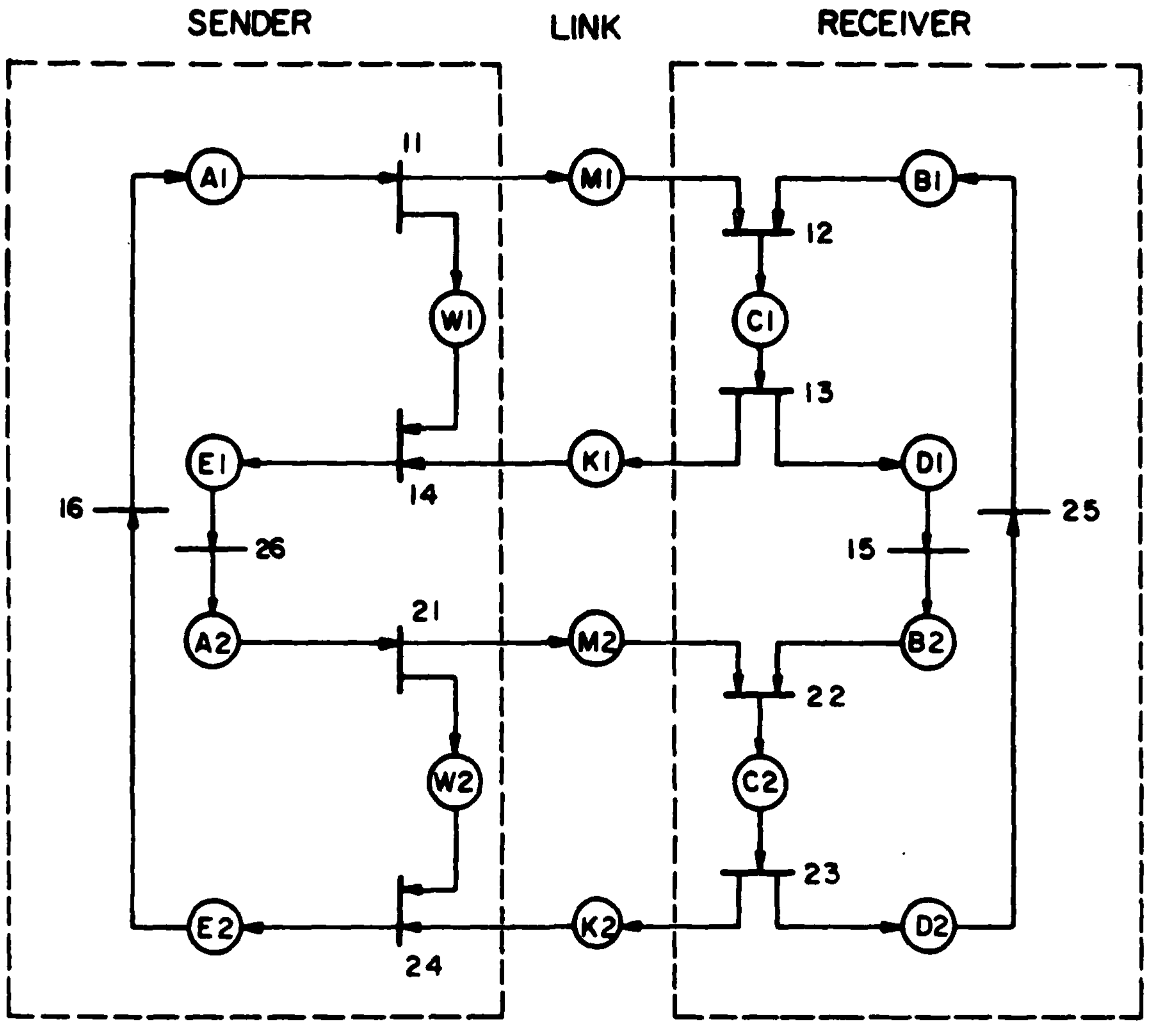

Figure 3.5. Alternating bit protocol as a Petri Net

"Pure" Petri nets suffer most of the same limitations as FSA, although they can represent an unbounded number of tokens (e.g., messages in transit). Hence a variety of extensions to Petri nets, such as inhibitor arcs, typed tokens, and state variables (or a separate "data graph"), have been proposed by various authors [Kell 76, GoMa 76, AABe 78, Symo 80, RaEs 80]. Petri nets extended in this fashion have an expressive power similar to abstract machines. It is somewhat easier to combine Petri nets into composite systems by identifying the input places of one net with the outputs of another. Since multiple control tokens are allowed in Petri nets, the result is a single Petri net with concurrently executing subparts.

Another extension not usually found in abstract machines is the addition of timing constraints to the transitions. These may be necessary to rule out undesirable behavior in some cases [MeFa 76].

Buffer Histories

Buffer histories are similar to formal languages and sequence expressions in their attempt to define the input/output behavior of the system without any mention of explicit states. The basic model is a *process* or active computing entity that has a set of *buffers* through which it exchanges items of specified types with the outside world. Several processes may be connected by these buffers. The major component of the specifications are relations between the items read from and written to *different* buffers by *different* processes.

The division of the interaction history into separate histories for each buffer is sometimes an advantage (e.g., in stating that the output bears some relation to the input), and sometimes a disadvantage because the relative ordering of items in different buffers must be considered (e.g., whether a Send request in a message data buffer is preceded by a Connect request in a call control buffer). In the latter case, some sort of time stamps must also be associated with the items to allow a relative ordering. Of course both the time stamps and the buffer histories are specification constructs, and need not appear in any implementation.

The major development of this type of specification has been in the Gypsy system [GoCo 78]. This system caters to cyclic processes in which the buffer relations are specified to hold at *blockage* points when a process tries to read from an empty buffer or write to a full one. An example Gypsy specification for a simple data transfer service is given in Figure 3.6.

```
BEGIN

TYPE msg = ...

PROCEDURE ProtocolService
   (Sent: INPUT BUFFER[1] OF msg, Delivered: OUTPUT BUFFER[0] OF msg)
   BEGIN
      BLOCK ALLFROM(Sent) = ALLTO(Delivered)
   END

END
```

Figure 3.6. Protocol service in Gypsy

Other workers have also made use of similar event histories in their specifications, notably Hailpern [HaOw 80] and Luckham and Karp [LuKa 79]. In these cases, the histories may be of more general events or actions performed, and need not correspond to items actually placed in buffers.

Although experience with these methods is still rather limited, they appear useful when the behavior being specified is conveniently expressed as a relationship between sequences of items (e.g., output equal input). When a more temporally "local" view is needed (e.g., whether the last input was a Connect request or not), an explicit state notion that summarizes the entire history seems more convenient [Suns 79].

Abstract Data Types

Abstract data types are an attempt to encapsulate data and the operations that manipulate it in a fashion similar to the way procedures encapsulate portions of program behavior. Two main approaches have emerged in this area: abstract model and axiomatic [Gutt 79].

The abstract model approach is essentially a formalization of the notion of abstract machine discussed above. However, the data types used as components of the state vector may themselves be (more primitive) user-defined abstract data types. The specifications give the effects of each operation of the new type on the set of data types that constitute the "model" (state vector) for the new type. For example, a stack might be defined by using a sequence as the model. The effect of one operation on a subsequent operation is given indirectly, by changing the state, which in turn determines the effect of the subsequent operation.

The axiomatic approach attempts to define the operations of a new type directly without recourse to an underlying model, by defining the effects of the operations on each other. In particular, there are *constructor* operations which change or create new objects of the type (e.g., NewStack, Push(stack,item)), and *selector* operations that reveal properties of the objects by returning values of other types (e.g. Empty(stack): boolean, or Size(stack): integer). There may also be *modifier* operations (e.g., Pop(stack)) which change objects of the new type, but whose effects may be defined in terms of the primary constructors.

The effects of operations are given as a set of *axioms* that define the effects of combinations of operations directly. The axioms may be interpreted as rewrite rules that show how to reduce any combination of operations to a simpler form. For example, Top(Push(stack,item)) = = item and Pop(Push(stack,item)) = = stack would be axioms for a stack.

The distinction between these two approaches may not be so great in practice [FlMi 79]. In particular, it is possible to write abstract model specifications in the axiomatic notation, as shown in Figure 3.7, which specifies a simple data transfer service using AFFIRM [Gerh 80, Thom 81]. Experience with these techniques is still limited, but their ability to formalize the widely used abstract machine model is very promising. The ability to incorporate other data types in a new type and the existence of some automated tools for checking specifications have proved very powerful aids. However, major problems remain in the area of exception handling, concurrency, and composition of systems from parts.

```
type ProtocolService;

needs types Message, QueueOfMessage;

declare p: ProtocolService;
declare m: Message;

interfaces  Sent(p), Received(p), Buf(p):  QueueOfMessage;

interfaces  Init, Send(p,m), Deliver(p): ProtocolService;

interface  Empty(p):  Boolean;


define Empty(p) == Buf(p)=NewQueueOfMessage;


axioms

   Sent(Init) == NewQueueOfMessage,
   Received(Init) == NewQueueOfMessage,
   Buf(Init) == NewQueueOfMessage,

   Sent(Send(p,m)) ==
     if not Empty(p) then Sent(p) Add m
     else Sent(p),
   Received(Send(p,m)) == Received(p),
   Buf(Send(p,m)) ==
     if not Empty(p) then Buf(p) Add m
     else Buf(p),

   Sent(Deliver(p)) == Sent(p),
   Received(Deliver(p)) ==
     if not Empty(p) then Received(p) Add Front(Buf(p))
     else Received(p),
   Buf(Deliver(p)) ==
     if not Empty(p) then Remove(Buf(p))
     else Buf(p);
```

Figure 3.7. Protocol service in AFFIRM

Exception handling in abstract data type models is still under investigation [GoTa 79]. Because of the strong typing of each operation, it is awkward to define operations that return an error indication as well as their normal type value (e.g., what is Top(NewStack)?). It is possible to specify a separate test operation which tells whether an operation would be successful, but this is also inconvenient. Hence systems are usually specified to "ignore" invalid requests.

Programs and Program Assertions

Programs are one of the oldest methods of describing the operation of a system. Some of the earliest protocol analysis work was based on program specifications [Boch 75, Sten 76]. Later work by Krogdahl generalized these programs to include a broader class of allowed behavior [Krog 78]. Special language constructs tailored to specifying concurrent protocol modules have been another line of development [VyVi 80]. But the major shortcoming of programs as specifications remains their lack of abstraction--it is difficult to separate the necessary behavior of the system from the incidental aspects of the program chosen to implement it.

Recent work incorporating temporal logic is aimed at specifying the necessary behavior of the system with minimal or no reference to explicit programs. The basic temporal operators *eventually* and *henceforth* can be added to traditional assertions about programs using predicate calculus. Combined with the notion that the locus of control is *at* (just before), *in*, or *after* a statement, temporal logic facilitates specifications of the behavior of systems over time. This makes specification of progress properties as well as safety properties of the system easier.

Hailpern and Owicki have developed specifications employing data abstraction, event history, and temporal logic techniques [HaOw 80]. Both active and passive components may be specified, corresponding to processes and abstract machines. For the passive components (called monitors), the effects of operations are only partly specified in the usual way for each operation, and other properties are given in invariants on the variables of the system (for safety) and in *commitments* using the temporal operators to specify liveness properties. Only invariants and commitments need be given for processes.

An example of an invariant might be *msg in output implies msg in input* as a description of a medium that can lose but not invent new messages. An example of a commitment might be *at Receive and MessageExists implies eventually after Receive*, indicating that a Receive operation will complete if a message is available.

Schwartz and Melliar-Smith have developed specifications employing temporal logic with a more explicit notion of system state [ScMe 81]. They claim that the use of temporal logic allows a more general description of the system than a state machine model. For example, incrementing a sequence number might be associated with several operations of a protocol machine and still allow correct behavior, but an abstract machine model must pick one. Their specifications allow them to state the minimum necessary condition that successive packets have different sequence numbers without having to specify just how this is accomplished. They have also experimented with a number of more specialized temporal operators such as *until* and *latches while*.

State deltas are another method making implicit use of temporal logic [Croc 77]. A state delta forms the basic unit of specifications, giving a precondition, modification list, and a postcondition.

The state delta says that if the precondition becomes true at some time t1, then eventually (at some time t2>t1) the postcondition will be true, and between t1 and t2 only the variables listed in the modification list may change.

State deltas were originally developed for sequential (single process) programs, but they are currently being extended for concurrent programs. Time constraints in basic state deltas, and a *Wait* primitive with a timeout have also been added [Over 80]. Translators from some program languages to state deltas have been written. Figure 3.8 gives a specification for the Alternating Bit protocol illustrating some of these features (with the mod lists omitted for brevity).

As suggested by the term state delta, the distinction between program and state machine specifications is not fundamental. If the program counter is viewed as the main state variable, then programs can be translated into an abstract machine model. In state deltas, the program counter typically forms part of the precondition, and appears in the postcondition with a new value (the next state). On the other hand, state machines can clearly be expressed as programs.

4. VERIFICATION

While clear specifications are valuable in their own right, a major motivation for more rigorous specifications is to enable formal verification as described in Section 2. Some verification techniques are closely coupled with particular specification methods, while others are more broadly applicable. In this section we present the major verification techniques, indicate where they have been used, and discuss their relative merits.

State Exploration

State exploration is a constructive technique for exploring the possible behavior of state machine systems. The method is based on building the graph of reachable states starting from a specified initial state. For each state in the graph, arcs to all possible immediate successor states are added. If the successors are either previously generated states (the graph has cycles) or terminal states (with no successors), then no further exploration of those nodes is needed. Clearly, the process will terminate if there are a finite number of states.

State exploration is directly applicable to checking finite state automata (FSA) specifications. It can identify a number of undesirable properties of a system including deadlocks (undesired terminal states), cycles (which may or may not be desired), incompleteness (the possibility of an event whose processing is not specified), and overspecification (portions of the specification that are never used).

In addition to checking single FSA, the method can be used to analyze systems of FSA connected by message queues. A *composite* or global system state is constructed including the states of each FSA and the values of the queues, and the exploration method is applied considering all possible

```
Sender Description

[PSD  pre:  S at 1
      read: INPUT
      post: #SMessage=.INPUT, S at 2]
        {# means new value of, . means old value of}

[PSD  pre:  S at 2
      time: SendDelay
      post: if UNDEFINED or .Slost ≥ MAXLOSS then S at 3T else S at 3F]
        {allows message to be lost up to MAXLOSS times}

[PSD  pre:  S at 3T  {successful transmission}
      post: #Sdata=.SMessage, #Sseq=.SendSeqNo, #SReadyflag=TRUE,
            #Slost=0, S at 4]

[PSD  pre:  S at 3F  {bad transmission}
      post: #Slost=.Slost+1, S at 4]

[WAIT pre:  S at 4  {wait for flag or timeout}
      exp:  .RReadyflag
      time: Timeout
      thenpost: S at 5  {if flag}
      elsepost: S at 2]  {if timeout}

[PSD  pre:   S at 5
      post: #RReadyflag=FALSE,
            (if .Rseq=.SendSeqNo then #SendSeqNo=¬.SendSeqNo, S at 1
                                    else  S at 2)]


Receiver Description

[WAIT pre:  R at 1
      exp:  .SReadyflag
      thenpost: #SReadyflag=FALSE, #ReceivedSeqNo=.Sseq,
                (if .ExpectedSeqNo=.Sseq then R at 2 else R at 3)]

[PSD  pre:  R at 2
      post: #OUTPUT=.Sdata, #ExpectedSeqNo=¬.ExpectedSeqNo, R at 3]

[PSD  pre:  R at 3
      time: AckDelay
      post: if UNDEFINED or .Rlost ≥ MAXLOSS then R at 4T else R at 4F]
        {allows Ack to be lost up to MAXLOSS times}

[PSD  pre:  R at 4T  {successful Ack}
      post: #Rseq=.ReceivedSeqNo, #RReadyflag=TRUE, #Rlost=0, R at 1]

[PSD  pre:  R at 4F  { bad Ack}
      post: #Rlost=.Rlost+1, R at 1]
```

Figure 3.8. Alternating bit protocol in state deltas

transitions of each FSA [West 78, Boch 78]. In this case, overflow of a specified maximum queue size can also be checked.

A similar form of analysis may be applied to Petri nets if the marking of the Petri net (the vector giving the number of tokens in each place) is viewed as the state. In this domain, the reachability graph is called a *computation flow graph* (CFG). A property called *proper termination* is equivalent to reaching a proper final state, and may be checked by a reduction process that is faster than construction of the CFG [Post 74, RaEs 80]. Faster analysis can also be accomplished by modeling properties of the net with linear algebra [AABe 78].

Most of the results to date with these methods have focused on the properties of a single level of specification. In order to show that the protocol meets a separate service specification, it is necessary to show the "equivalence" of two state systems. We shall say more about this problem below.

As seen from the above discussion, a rich theory and variety of tools have been developed for this type of analysis. It is also easily automated, and a number of useful results have been obtained, for example with the CCITT X.21 protocol [WeZa 78, RaEs 80]. Unfortunately, the limitations of FSA and pure Petri net specifications mean that these tools can only be used on simple protocols, or on the control aspects of more complex protocols.

Symbolic Execution

Symbolic execution is an attempt to carry over the basic method of state exploration to richer types of specification. Although the term suggests a program type specification, it is equally applicable to abstract machine specifications. The goal is still to construct a graph of reachable system states, but now the state is represented "symbolically," and may correspond to a large class of particular systems states (e.g., all states with a single message pending, no matter what its contents).

Again starting from the initial composite system state, the reachability graph is constructed, making branches when a "more symbolic" state must be broken into two possible "more concrete" cases (e.g., when an arriving message has a sequence number equal or not equal to some symbolic value). This drastically reduces the size of the reachability graph, but complicates correspondingly the test for whether newly generated states are identical to previous states, since it requires comparison of possibly complex symbolic state expressions. As with state exploration, branches are also constructed to represent the various possible interleavings of the concurrently executing components of the system.

A minimally symbolic execution of several protocols has been performed by Hajek [Haje 78]. Brand and Joyner have experimented with more symbolic analysis of a simple protocol [BrJo 78] and HDLC [BrJo 79]. Bremer and Drobnik used a similar technique on the HDLC protocol [BrDr 79]. State deltas can also be symbolically executed, with the goal of demonstrating that a lower level specification properly implements a higher level specification, although this work has only begun to be applied to protocols [Over 80]. All three of these examples used automated systems, while Sunshine performed an early manual symbolic execution of a connection establishment procedure [SuDa 78], and Bochmann more recently completed analysis of a hierarchy of protocols [Boch 79].

The more powerful of these techniques are far from automatic, and require input of intermediate assertions, induction hypotheses, and other guidance from human users. Hence the increased power over simple state exploration systems comes at a definite price.

Structural Induction

The basic proof method used with abstract data types is *structural induction*. As noted in Section 3, there are a specified set of primary operations that create or modify objects of a data type. If we wish to prove that some property P holds for all objects of the type (i.e., P is an *invariant*), it suffices to show (1) that P holds for all the directly created objects (the *basis*), and (2) that if P holds for an object, then it still holds after every constructor which can modify objects of the type has been applied (the *induction step*) [Gutt 78].

Using this kind of induction and the ordinary rules of logic along with the specifications of the operations, it is possible to prove useful invariants for a type, for example, that the output messages are always an initial subsequence of the input messages [Thom 81]. Usually these properties concern safety, but they may also concern liveness, for example, that it is always true that either some operation is enabled or the system is in a correct final state [Bert 80].

Ina Jo has structural induction built into its specification checker so that type specifications and invariants are read in together, and for each operation a theorem is produced that must be verified. In AFFIRM, specifications may be read in alone. Proposed invariants may be added at a later time, and the user must explicitly employ induction if desired in carrying out the proof [Gerh 80]. Both systems have an interactive prover that the user must guide through proofs. The proof process is typically an iterative one where difficulties in the proof suggest changes to the specifications.

The problem of proving that a lower level specification correctly "implements" a higher level specification is relatively well understood for abstract data types [Gutt 78]. Essentially, this involves showing that the objects of the implementing type represent all the objects of the implemented type somehow, and that each operation of the implemented type is performed by some sequence of operations in the implementing type that has the same effects. To accomplish the latter, the effects of the higher level operations are basically "mapped down" into corresponding assertions about effects of implementing level operations which are then proved from the implementing level specifications. This often relies on some key system invariant that is proved by induction.

A special problem in showing this correspondence between protocol and service specifications is that one service level operation may be implemented by a nondeterministic sequence of protocol level operations. For example, the user level Send message operation will be accomplished by some unknown combination of (Re)transmit, Receive, and Acknowledge operations at the protocol level if messages can be lost in transit. Solutions to this problem are now being explored with AFFIRM [Thom 81].

Program Verification

Early work in this area applied Hoare's assertion type methods to protocols specified procedurally as programs [Boch 75, Sten 76]. In this approach, assertions must be attached at suitable points in the programs. Higher level assertions must also be formulated, serving as a partial service specification. This often requires the introduction of "ghost variables" (which are not implemented) to capture the desired notion of correct behavior. For example, arrays of data sent and delivered may be needed to specify the desired behavior of a data transfer protocol, much as with buffer histories.

The low-level assertions must then be verified from the programs, and are in turn used to verify the higher level assertions. A good deal of ingenuity is required both in formulating appropriate assertions and in carrying out the proofs. The process is typically an iterative one where difficulties in the proof suggest changes to the assertions. Needless to say, the fact that protocol systems have several components running concurrently complicates the problem.

More recent specifications have become more abstract and less procedural. We have already mentioned the blockage assertions and buffer histories that are used to characterize programs in Gypsy. The introduction of temporal logic into assertions also allows greater generality by broadening the future time at which a condition will hold (e.g., "eventually"). However, the basic proof method of characterizing the behavior of a program by a set of assertions, and then proving that the combined operation of several such components satisfies some higher level assertions, remains unchanged. There has been less progress in automating this method than with the others described here, although some promising manual proofs have been performed [HaOw 80].

Design Rules

The techniques falling into this category do not attempt to analyze already completed specifications at all, but rather to prescribe rules for the construction of specifications that will guarantee their correctness. One method is aimed at adding all necessary receive transitions whenever a send transition is specified [Zafi 80, BrZa 80]. For example, if a send event is added from a state where a receive is specified, then the receive must also be specified in the state reached after the send.

Another method takes a specification of the desired service and of all but one submodule intended to implement the service, and constructs the missing module needed (if one exists) [BoMe 80]. The first method uses finite state specifications while the second uses essentially equivalent grammar models, so that the complexity of protocols that can be synthesized has been limited.

5. CONCLUSIONS

A variety of techniques have been applied to the specification and analysis of computer communication protocols. For the most part these techniques were developed earlier for more general problems, and the existing theory and analysis procedures have been carried over to the protocol area.

Simple methods such as finite state automata and Petri nets have yielded surprisingly useful results [WeZa 78, Schu 80, RaEs 80]. Such specifications are fairly easy to understand, and automated analysis is relatively straightforward. Hence this technology appears to be ready for more widespread use.

Although such simple techniques have a well-developed theory, they have proved inadequate for fully describing complex real-world protocols. Numerous extensions have been developed to improve their expressive power, and several major applications of these more powerful methods have been successfully demonstrated (see [BoSu 80] for a list and brief discussion). Here we only mention [BoCh 77, Boch 80, SFAI 80, HaOw 80] as illustrating some of the more advanced specification methods.

Abstract machines seem to be the most popular extended model, sharing with simpler models their relative ease of understanding. Analysis is more difficult, but some promising results are discussed in [Thom 81]. Several significant applications indicate that this technique also appears ready for more widespread use [Dick 80, ACFa 79, Thom 81].

However, the need to invent and manipulate an explicit state may be seen as a disadvantage of abstract machine models, leading to consideration of sequencing expression or buffer history models which avoid explicit state notions. Temporal logic methods aim to alleviate other difficulties with liveness properties and the passive nature of abstract machines. These methods have promising advantages in particular areas, but generally are in more exploratory stages of development.

Unfortunately, many of these promising extensions to simple specification techniques invalidate or complicate the original analysis methods. Our analysis abilities have not yet caught up with our specification abilities. Although there has also been notable progress in this area, much work remains to be done before complete protocol verification becomes a routine part of the design process.

REFERENCES

[AABe 78] Azema, P., J. M. Ayache, and B. Berthomieu, "Design and verification of communication procedures: A bottom-up approach," *Proceedings of the Third International Conference on Software Engineering*, 1978.

[ACFa 79] Alfonzetti, S., S. Casale, and A. Faro, "A formal description of the DTE packet level in the X.25 recommendation," *Alta Frequenza* 48, 8, August 1979, pp. 339 E-513 - 340 E-514.

[BaRa 80] Bartlett, K., and D. Rayner, "The certification of data communication protocols," *Proceedings of the Trends and Applications Symposium*, National Bureau of Standards (USA), May 1980.

[Bert 80] Berthomieu, B., Proving liveness properties of communication protocols in AFFIRM, USC/Information Sciences Institute, AFFIRM Memo 35, September 1980.

[Boch 75] Bochmann, G. V., "Logical verification and implementation of protocols," *Proceedings of the 4th Data Communications Symposium*, Quebec, 1975, pp. 8-15 to 8-20.

[BoCh 77] Bochmann, G. V., and R. J. Chung, "A formalized specification of HDLC classes of procedures," *Proceedings of the National Telecommunications Conference*, Los Angeles, December 1977, paper 3A.2.

[Boch 78] Bochmann, G. V., "Finite state description of communication protocols," *Computer Networks* 2, 4/5, October 1978, pp. 361-372.

[Boch 79] Bochmann, G. V., *Formalized Specification of the MLP, Specification of the Services Provided by the MLP, and An Analysis of the MLP*, University of Montreal, Department d'I.R.O., 1979.

[Boch 80] Bochmann, G. V., "A general transition model for protocols and communication services," *IEEE Transactions on Communications* COM-28, 4, April 1980, pp. 643-650.

[BoMe 80] Bochmann, G. V., and P. Merlin, "On the construction of communication protocols," *Proceedings of the International Conference on Computer Communication*, Atlanta, October 1980, pp. 371-378.

[BoSu 80] Bochmann, G. V., and C. A. Sunshine, "Formal methods in communication protocol design," *IEEE Transactions on Communications* COM-28, 4, April 1980, pp. 624-631.

[BrDr 79] Bremer, J., and O. Drobnik, *A New Approach to Protocol Design and Validation*, IBM Research Report RC 8018, December 1979.

[BrJo 78] Brand, D., and W. H. Joyner, Jr., "Verification of protocols using symbolic execution," *Computer Networks* 2, 4/5, October 1978, pp. 351-360.

[BrJo 79] Brand, D., and W. H. Joyner, Jr., *Verification of HDLC*, IBM Research Report RC 7779, July 1979.

[BrZa 80] Brand, D., and P. Zafiropulo, "Synthesis of protocols for an unlimited number of processes," *Proceedings of the Trends and Applications Symposium*, National Bureau of Standards (USA), May 1980.

[Croc 77] Crocker, S., *State Deltas: A Formalism for Representing Segments of Computation*, Ph.D. thesis, University of California, Los Angeles, 1977.

[DaBr 78] Danthine, A., and J. Bremer, "Modelling and verification of end-to-end transport protocols," *Computer Networks* 2, 4/5, October 1978, pp. 381-395.

[Dick 80] Dickson, G. J., "Formal specification technique for data communication protocol X.25 using processing state transition diagrams," *Australian Telecommunication Research* 14, 2, 1980.

[FlMi 79] Flon, L., and J. Misra, "A unified approach to the specification and verification of abstract data types," *Proceedings of the Conference on Specification of Reliable Software*, 1979, pp. 162-169.

[Gerh 80] Gerhart, S. L., et al., "An overview of AFFIRM: A specification and verification system," *Proceedings of the IFIP Congress*, October 1980, pp. 343-348.

[GoCo 78] Good, D., and R. M. Cohen, "Verifiable communications processing in Gypsy," *Proceedings of the 17th IEEE Computer Society International Conference (COMPCON)*, September 1978, pp. 28-35.

[GoMa 76] Gouda, M. G., and E. G. Manning, "On the modelling, analysis, and design of protocols--A special class of software structures," *Proceedings of the 2nd International Conference on Software Engineering*, October 1976, pp. 256-262.

[GoTa 79] Goguen, J. A., and J. J. Tardo, "An introduction to OBJ," *Proceedings of the Conference on Specification of Reliable Software*, 1979, pp. 170-189.

[Gutt 78] Guttag, J. V., E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Communications of the ACM* 21, 12, December 1978, pp. 1048-1064.

[Gutt 79] Guttag, J., "Notes on type abstraction," *Proceedings of the Conference on Specification of Reliable Software*, 1979, pp. 36-46.

[Haje 78] Hajek, J., "Automatically verified data transfer protocols," *Proceedings of the 4th International Computer Communication Conference*, Kyoto, September 1978, pp. 749-756.

[HaOw 80] Hailpern, B., and S. Owicki, "Verifying network protocols using temporal logic," *Proceedings of the Trends and Applications Symposium*, National Bureau of Standards (USA), May 1980.

[Hara 77] Harangozo, J., "An approach to describing a link level protocol with a formal language," *Proceedings of the 5th Data Communication Symposium*, Snowbird, Utah, 1977, pp. 4-37 to 4-49.

[Kell 76] Keller, R. M., "Formal verification of parallel programs," *Communications of the ACM* 19, 7, July 1976, pp. 371-384.

[Krog 78] Krogdahl, S., "Verification of a class of link-level protocols," *BIT* 18, 1978, pp. 436-448.

[LeMo 73] LeMoli, G., "A theory of colloquies," *Alta Frequenza* 42, 10, 1973, pp. 493-223E to 500-230E; and *Proceedings of the First European Workshop on Computer Networks*, Arles, April 1973, pp. 153-173.

[Loca 80] Locasso, R., et al., *The Ina Jo Specification Language Reference Manual*, System Development Corp. TM-(L)-6021/001/00, June 1980.

[LuKa 79] Luckham, D. C., and R. A. Karp, *An Axiomatic Semantics of Concurrent Cyclic Processes*, Stanford University Artificial Intelligence Laboratory, 1979.

[Merl 79] Merlin, P. M., "Specification and validation of protocols," *IEEE Transactions on Communications*. COM-27, 11, November 1979, pp. 1671-1680.

[MeFa 76] Merlin, P. M., and D. J. Farber, "Recoverability of communication protocols--Implications of a theoretical study," *IEEE Transactions on Communications*, September 1976, pp. 1036-1043.

[Over 80] Overman, W., Verification of concurrent systems: function and timing, USC/Information Sciences Institute (in preparation).

[Parn 72] Parnas, D. L., "A technique for software module specification with examples," *Communications of the ACM* 15, 5, May 1972, pp. 330-336.

[Piat 80] Piatkowski, T., "Remarks on ADCCP validation and testing techniques," *Proceedings of the Trends and Applications Symposium*, National Bureau of Standards (USA), May 1980.

[Post 74] Postel, J. B., *A Graph Model Analysis of Computer Communications Protocols*, Ph.D. thesis, University of California, Los Angeles, 1974.

[RaEs 80] Razouk, R., and G. Estrin, "Validation of the X.21 interface specification using Sara," *Proceedings of the Trends and Applications Symposium*, National Bureau of Standards (USA), May 1980.

[RLSi 79] Robinson, L., K. N. Levitt, and B. A. Silverberg, *The HDM handbook*, 3 volumes, SRI International, June 1979.

[Schi 80] Schindler, S., "Algebraic and model specification techniques," *Proceedings of the 13th Hawaii International Conference on System Sciences*, January 1980.

[Schu 80] Schultz, G. D., et al., "Executable description and validation of SNA," *IEEE Transactions on Communications* COM-28, 4, April 1980, pp. 661-677.

[ScMe 81] Schwartz, R. L., and P. M. Melliar-Smith, "Temporal logic specification of distributed systems," *Proceedings of the Second International Conference on Distributed Computing Systems*, IEEE, 1981.

[SDCo 80] System Development Corp., "Protocol specification technique," in *Formal Description Techniques for Network Protocols*, Report No. ICST/HLNP 80-3, National Bureau of Standards (USA), June 1980.

[SFAl 80] Schindler, S., U. Flasche, and D. Altenkruger, "The OSA project: Formal specification of the ISO transport service," *Proceedings of the Computer Networking Symposium*, National Bureau of Standards (USA), December 1980.

[Sten 76] Stenning, N. V., "A data transfer protocol," *Computer Networks* 1, 2, September 1976, pp. 99-110.

[SuDa 78] Sunshine, C. A., and Y. K. Dalal, "Connection management in transport protocols," *Computer Networks* 2, 6, December 1978, pp. 454-473.

[Suns 79] Sunshine, C. A., *Formal Methods for Communication Protocol Specification and Verification*, N-1429, The Rand Corporation, November 1979.

[Symo 80] Symons, F. J. W., *Representation, Analysis, and Verification of Communication Protocols*, Telecom Australia Research Labs Report No. 7380, 1980.

[TeLi 78] Teng, A. Y., and M. T. Liu, "A formal model for automatic implementation and logical validation of network communication protocols," *Proceedings of the Computer Networking Symposium*, National Bureau of Standards (USA), December 1978, pp. 114-123.

[Tenn 80] Tenney, R., "Specification technique," in *Formal Description Techniques for Network Protocols*, Report No. ICST/HLNP 80-3, National Bureau of Standards (USA), June 1980.

[Thom 81] Thompson, D., et al., *Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models*, USC/Information Sciences Institute, Research Report 81-88, March 1981.

[VyVi 80] Vytopil, J., and C. Vissers, *Interaction Primitives in Formal Specification of Distributed Systems*, Twente University, The Netherlands, June 1980.

[West 78] West, C. H., "General technique for communication protocol validation," *IBM Journal of Research and Development*, 22, 4, July 1978.

[WeZa 78] West, C. H., and P. Zafiropulo, "Automated validation of a communications protocol: the CCITT X.21 recommendations," *IBM Journal of Research and Development*, 22, 1, January 1978, pp. 60-71.

[Zafi 80] Zafiropulo, P., et al., "Towards analyzing and synthesizing protocols," *IEEE Transactions on Communications* COM-28, 4, April 1980, pp. 651-661.

LMED

8